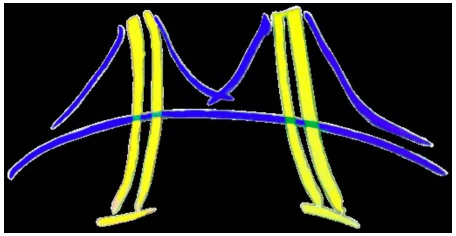
Programming Distributed Memory Systems with MPI

Tim Mattson

Intel Labs.



With content from Kathy Yelick, Jim Demmel, Kurt Keutzer (CS194) and others in the UCB EECS community. www.cs.berkeley.edu/~yelick/cs194f07,

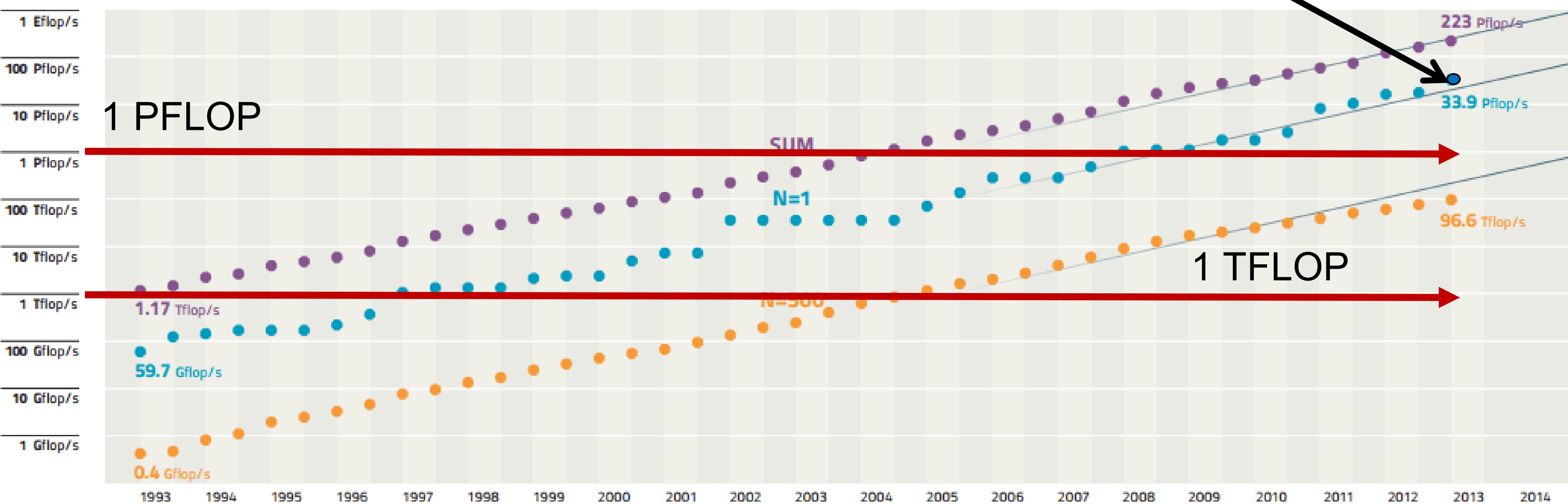
Outline

- Distributed memory systems: the evolution of HPC hardware
- Programming distributed memory systems with MPI
  - MPI introduction and core elements
  - Message passing details
  - Collective operations
- Closing comments

Tracking Supercomputers: Top500

- Top500: a list of the 500 fastest computers in the world (www.top500.org)
- Computers ranked by solution to the MPLinpack benchmark:
  - Solve Ax=b problem for any order of A
- List released twice per year: in June and November

Current number 1 (June 2013) 33.9 PFLOPS
Tianhe-2, NUDT, Intel Ivy Bridge + Xeon Phi
16.3 PFLOPS, >1.5 million cores



Source: http://s.top500.org/static/lists/2013/06/TOP500_201306_Poster.pdf

The birth of Supercomputing

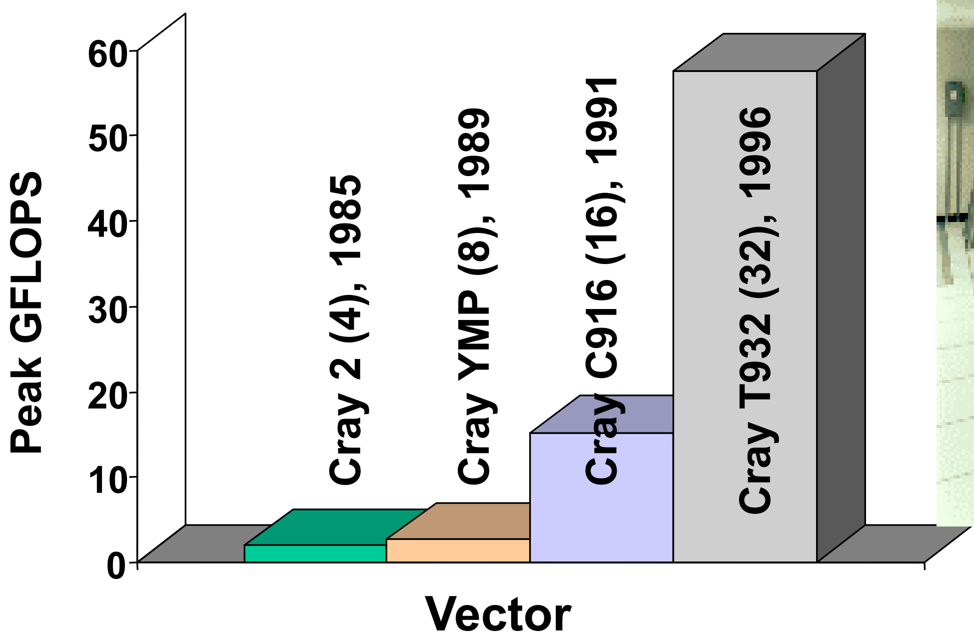
- On July 11, 1977, the CRAY-1A, serial number 3, was delivered to NCAR. The system cost was $8.86 million ($7.9 million plus $1 million for the disks).

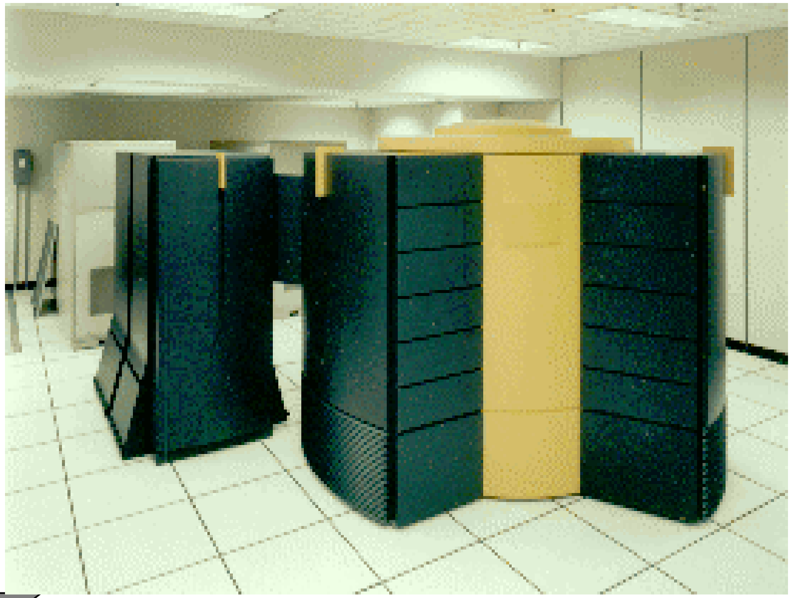
http://www.cisl.ucar.edu/computers/gallery/cray/cray1.jsp

- The CRAY-1A:
  - 2.5-nanosecond clock,
  - 64 vector registers,
  - 1 million 64-bit words of high-speed memory.
  - Peak speed:
    - 80 MFLOPS scalar.
    - 250 MFLOPS vector (but this was VERY hard to achieve)
- Cray software … by 1978
  - Cray Operating System (COS),
  - the first automatically vectorizing Fortran compiler (CFT),
  - Cray Assembler Language (CAL) were introduced.

History of Supercomputing:

- Large mainframes that operated on vectors of data
- Custom built, highly specialized hardware and software
- Multiple processors in an shared memory configuration
- Required modest changes to software (vectorization)





The Cray C916/512 at the Pittsburgh Supercomputer Center

The attack of the killer micros

- The **Caltech Cosmic Cube** developed by Charles Seitz and Geoffrey Fox in1981
- 64 Intel 8086/8087 processors
- 128kB of memory per processor
- 6-dimensional hypercube network

The cosmic cube, Charles Seitz
Communications of the ACM, Vol 28, number 1 January 1985, p. 22

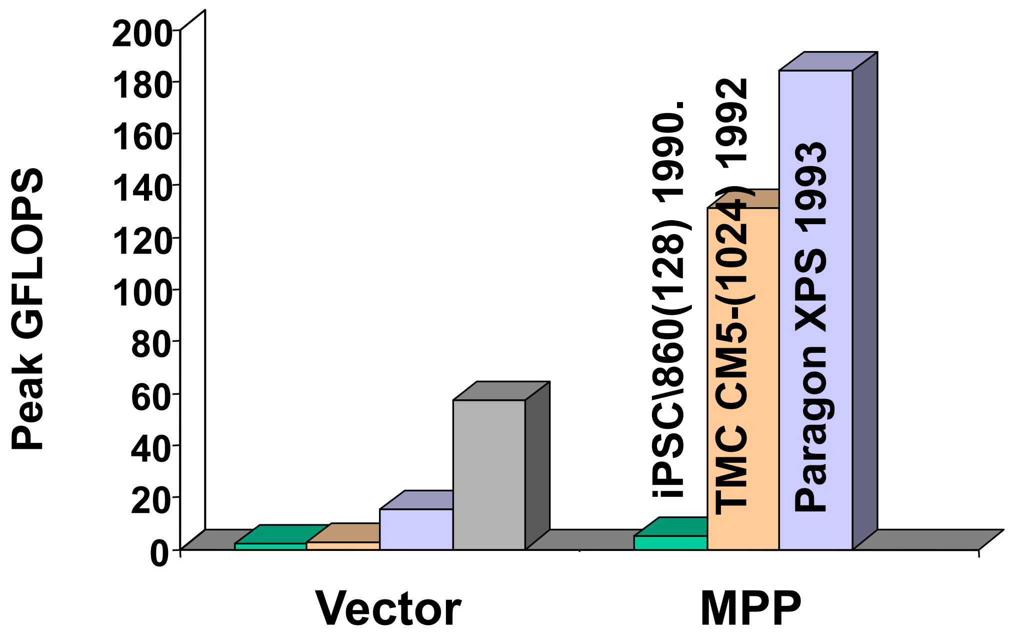
Launched the "attack of the killer micros"

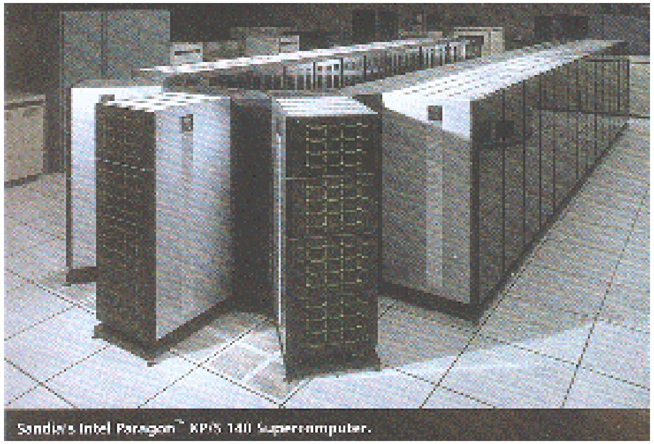
Eugene Brooks, SC'90

http://calteches.library.caltech.edu/3419/1/Cubism.pdf

It took a while, but MPPs came to dominate supercomputing

- Parallel computers with large numbers of microprocessors
- High speed, low latency, scalable interconnection networks
- Lots of custom hardware to support scalability
- Required massive changes to software (parallelization)

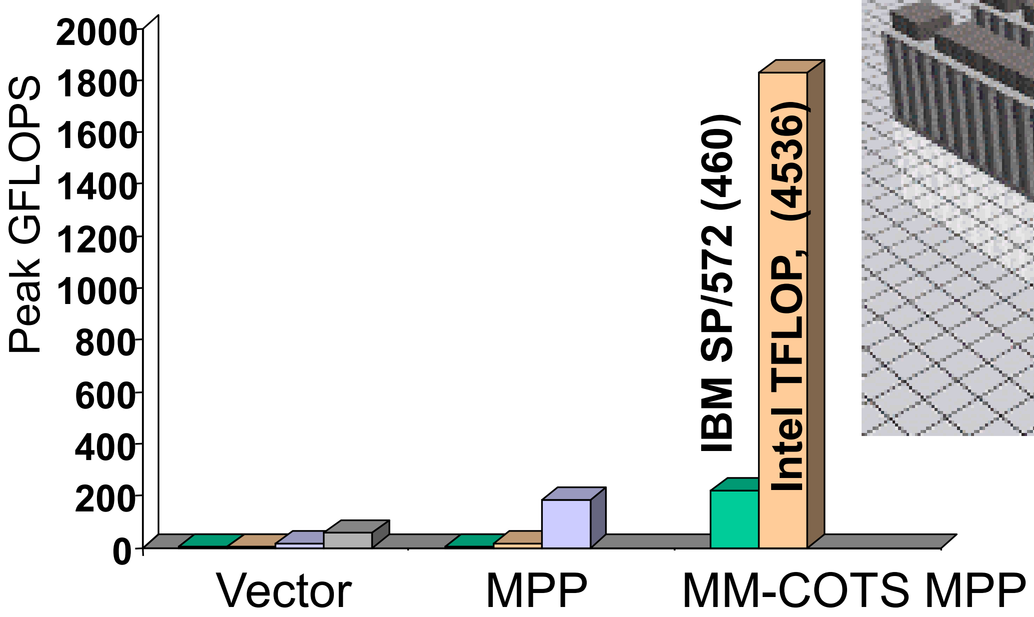


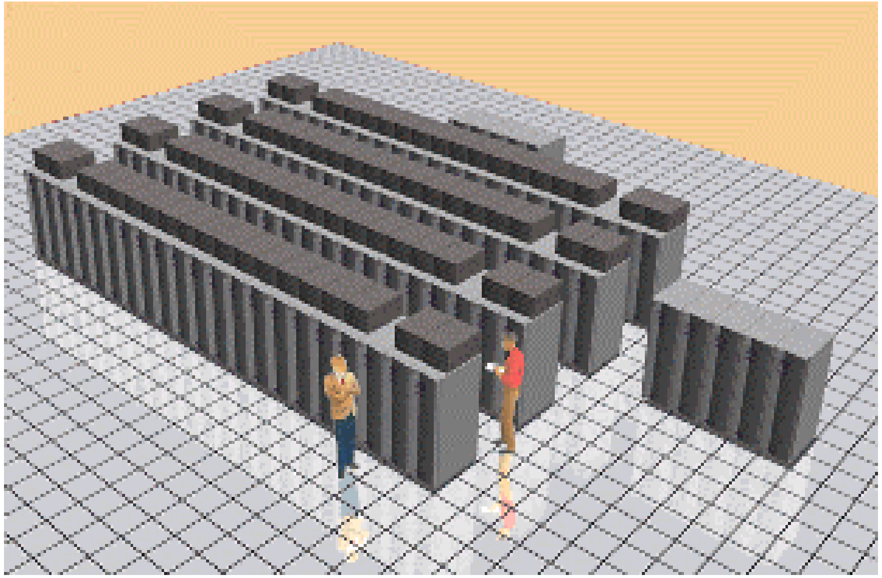


Paragon XPS-140 at Sandia National labs in Albuquerque NM

The cost advantage of mass market COTS

- MPPs using Mass market Commercial off the shelf (COTS) microprocessors and standard memory and I/O components
- Decreased hardware and software costs makes huge systems affordable



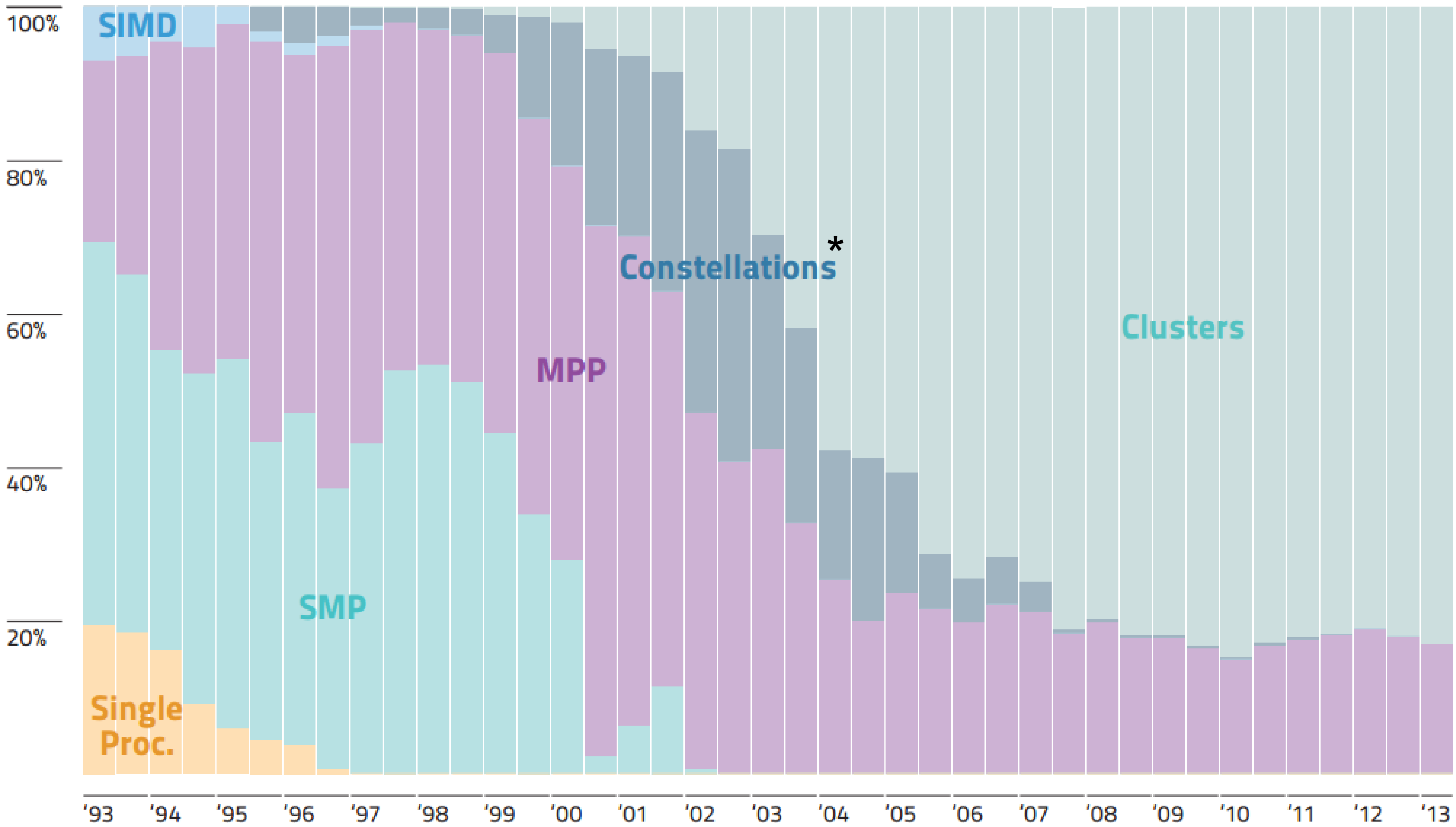


ASCI Red TFLOP Supercomputer

The MPP future looked bright … but then clusters took over

- A cluster is a collection of connected, independent computers that work in unison to solve a problem.
- Nothing is custom … motivated users could build cluster on their own
- First clusters appeared in the late 80’s (Stacks of “SPARC pizza boxes”)
- The Intel Pentium Pro in 1995 coupled with Linux made them competitive.
  - NASA Goddard’s Beowulf cluster demonstrated publically that high visibility science could be done on clusters.
- Clusters made it easier to bring the benefits due to Moores’s law into working supercomputers

Top 500 list: System Architecture



Source: http://s.top500.org/static/lists/2013/06/TOP500_201306_Poster.pdf

*Constellation: A cluster for which the number of processors on a node is greater than the number of nodes in the cluster. I've never seen anyone use this term outside of the top500 list.

Outline

- Distributed memory systems: the evolution of HPC hardware
- Programming distributed memory systems with MPI
  - MPI introduction and core elements
  - Message passing details
  - Collective operations
- Closing comments

MPI (1992-today)

- The message passing interface (MPI) is a standard library
- MPI Forum first met April 1992,
  - MPI 1.0 in June 1994
  - MPI 2.0 in July 1997
  - MPI 3.0 in September 2012
- Hardware-portable, multi-language communication library
- Enabled billions of dollars of applications
- Work on MPI 3.1 and 4.0 is in progress.


MPI Forum, March 2008, Chicago

MPI Hello World

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                            rank, size );
    MPI_Finalize();
    return 0;
}
```

Initializing and finalizing MPI

`int MPI_Init (int* argc, char* argv[])`

- Initializes the MPI library … called before any other MPI functions.
- agrc and argv are the command line args passed from main()

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                                    rank, size );
    MPI_Finalize();
    return 0;
}
```

`int MPI_Finalize (void)`

- Frees memory allocated by the MPI library … close every MPI program with a call to MPI_Finalize

How many processes are involved?

```
int MPI_Comm_size (MPI_Comm comm, int* size)
```

- **`MPI_Comm`**, an *opaque data type,* **a** communication context. Default context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_size`** returns the number of processes in the process group associated with the communicator

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                                   rank, size );
    MPI_Finalize();
    return 0;
}
```

Communicators consist of two parts, a **context** and a **process group**.

The communicator lets me control how groups of messages interact.

The communicator lets me write modular SW … i.e. I can give a library module its own communicator and know that it's messages can't collide with messages originating from outside the module

Which process "am I" (the rank)

```
int MPI_Comm_rank (MPI_Comm comm, int* rank)
```

- **`MPI_Comm`**, an *opaque data type,* **a** communication context. Default context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_rank`** An integer ranging from 0 to "(num of procs)-1"

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                                        rank, size );
    MPI_Finalize();
    return 0;
}
```

Note that other than init() and finalize(), every MPI function has a communicator.

This makes sense .. You need a context and group of processes that the MPI functions impact … and those come from the communicator.

Running the program

- On a 4 node cluster with MPIch2, I'd run this program (hello) as:

  > mpicc hello.c –o hello

  > mpiexec –n 4 –f hostf hello

  Hello from process 1 of 4

  Hello from process 2 of 4

  Hello from process 0 of 4

  Hello from process 3 of 4

- Where "hostf" is a file with the names of the cluster nodes, one to a line.

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                                  rank, size );
    MPI_Finalize();
    return 0;
}
```

Sending and Receiving Data

```
int MPI_Send (void* buf, int count,
        MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm)

int MPI_Recv (void* buf, int count,
        MPI_Datatype datatype, int source,
        int tag, MPI_Comm comm,
        MPI_Status* status)
```

- **`MPI_Send`** performs a blocking send of the specified data ("count" copies of type "datatype," stored in "buf") to the specified destination (rank "dest" within communicator "comm"), with message ID "tag"

- **`MPI_Recv`** performs a blocking receive of specified data from specified source whose parameters match the send; information about transfer is stored in "status"

By "blocking" we mean the functions return as soon as the buffer, "buf", can be safely used.

The data in a message: datatypes

- The data in a message to send or receive is described by a triple:
  - **(address, count, datatype)**
- An MPI datatype is recursively defined as:
  - Predefined, simple data type from the language (e.g., MPI_DOUBLE)
  - Complex data types (contiguous blocks or even custom types).
- E.g. … A particle's state is defined by its 3 coordinates and 3 velocities

```
MPI_Datatype PART;
MPI_Type_contiguous( 6, MPI_DOUBLE, &PART );
MPI_Type_commit( &PART );
```

- You can use this data type in MPI functions, for example, to send data for a single particle:

```
MPI_Send (buff, 1, PART, Dest, tag, MPI_COMM_WORLD);
```

address count Datatype

Receiving the right message

- The receiving process identifies messages with the double :
  - **(source, tag)**
- Where:
  - Source is the rank of the sending process
  - Tag is a user-defined integer to help the receiver keep track of different messages from a single source

MPI_Recv (buff, 1, PART, Src, tag, MPI_COMM_WORLD, &status);

Source tag

- Can relax tag checking by specifying MPI_ANY_TAG as the tag in a receive.
- Can relax source checking by specifying MPI_ANY_SOURCE

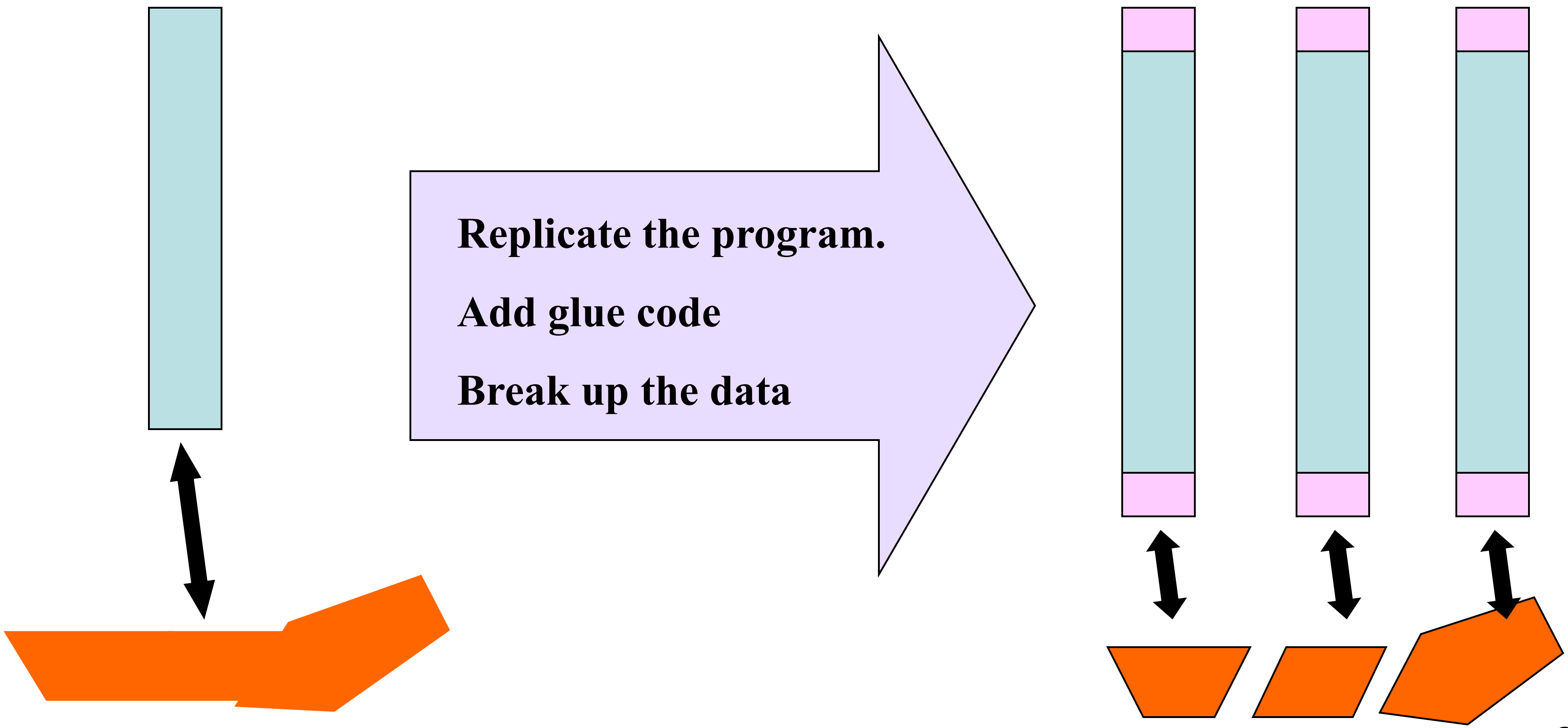
MPI_Recv (buff, 1, PART, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

- This is a useful way to insert race conditions into an MPI program

How do people use MPI? The SPMD Design Pattern

A sequential program working on a data set

- A single program working on a decomposed data set.
- Use Node ID and numb of nodes to split up work between processes
- Coordination by passing messages.



A Simple MPI Program

```
#include “mpi.h”
#include <stdio.h>
int main( int argc, char *argv[])
{ int rank, buf;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  /* Process 0 sends and Process 1 receives */
  if (rank == 0) {
    buf = 123456;
    MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              &status );
    printf( “Received %d\n”, buf );
  }
  MPI_Finalize();
  return 0;
}
```

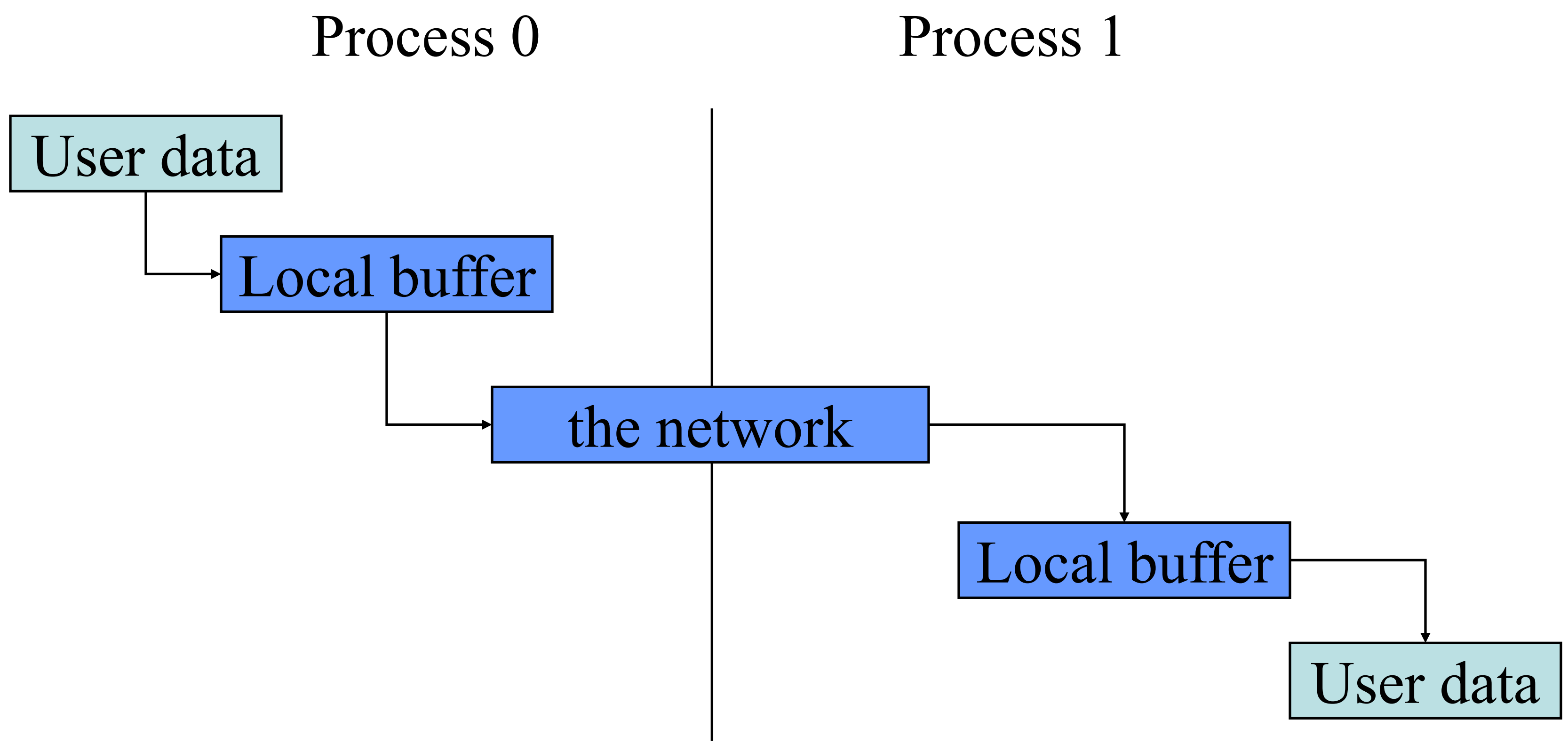
Slide source: Bill Gropp, ANL

Outline

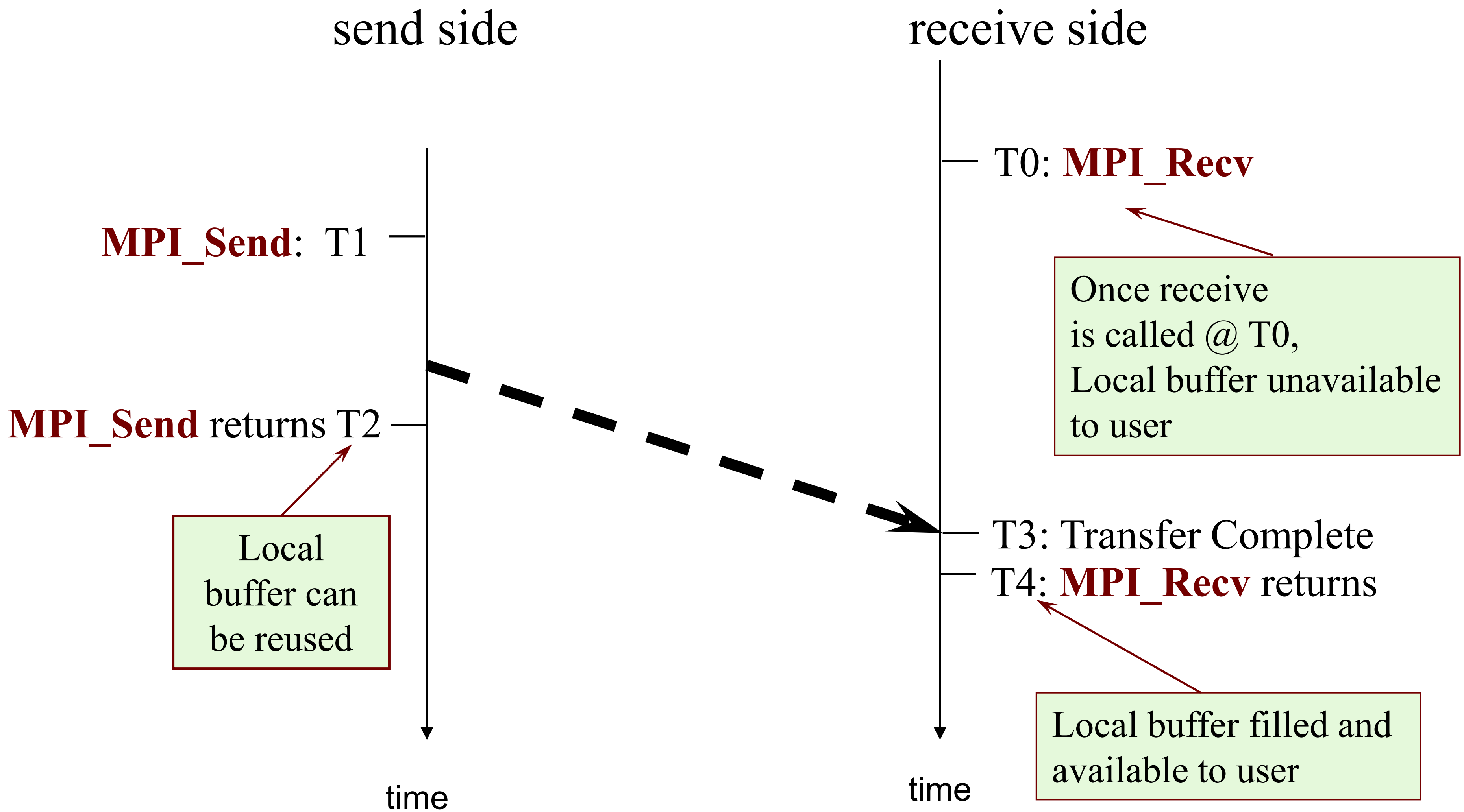
- Distributed memory systems: the evolution of HPC hardware
- Programming distributed memory systems with MPI
  - MPI introduction and core elements
  - Message passing details
  - Collective operations
- Closing comments

Buffers

- Message passing has a small set of primitives, but there are subtleties
  - Buffering and deadlock
  - Deterministic execution
  - Performance
- When you send data, where does it go? One possibility is:



Blocking Send-Receive Timing Diagram
(Receive before Send)



It is important to post the receive before sending, for highest performance.

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
`Send(1)`	**`Send(0)`**
`Recv(1)`	**`Recv(0)`**

- This code could deadlock … it depends on the availability of system buffers in which to store the data sent until it can be received

Slide source: based on slides from Bill Gropp, UIUC

Some Solutions to the "deadlock" Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	**`Recv(0)`**
`Recv(1)`	**`Send(0)`**

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	**`Sendrecv(0)`**

Slide source: Bill Gropp, UIUC

More Solutions to the "unsafe" Problem

- Supply a sufficiently large buffer in the send function

Process 0	Process 1
`Bsend(1)`	**`Bsend(0)`**
`Recv(1)`	**`Recv(0)`**

- Use non-blocking operations:

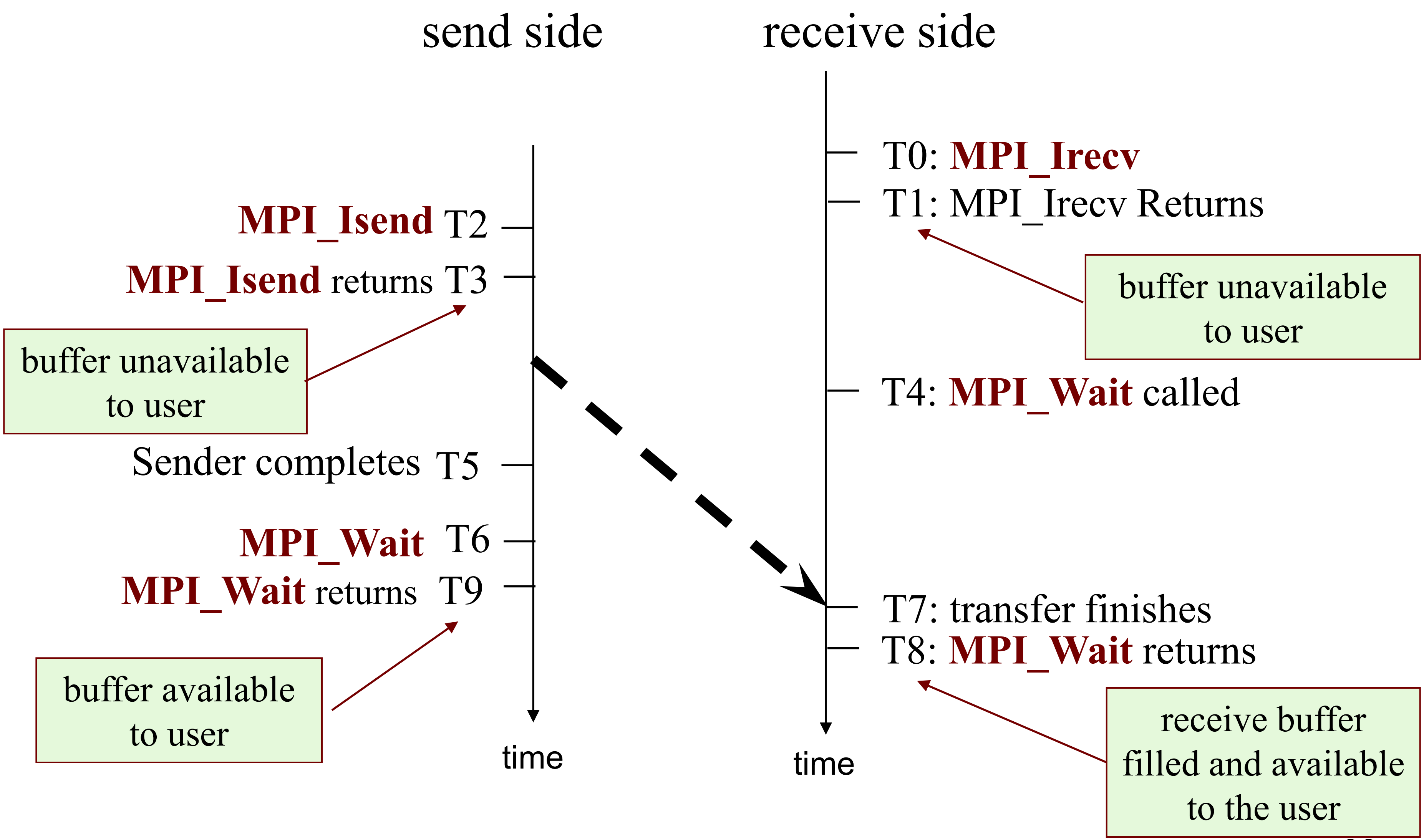
Process 0	Process 1
`Isend(1)`	**`Isend(0)`**
`Irecv(1)`	**`Irecv(0)`**
`Waitall`	**`Waitall`**

Non-Blocking Communication

- Non-blocking operations return immediately and pass "request handles" that can be waited on and queried
  - **MPI_ISEND(start, count, datatype, dest, tag, comm, request)**
  - **MPI_IRECV(start, count, datatype, src, tag, comm, request)**
  - **MPI_WAIT(request, status)**
- One can also test without waiting using MPI_TEST
  - **MPI_TEST(request, flag, status)**
- Anywhere you use MPI_Send or MPI_Recv, you can use the pair of MPI_Isend/MPI_Wait or MPI_Irecv/MPI_Wait

Non-blocking operations are extremely important … they allow you to overlap computation and communication.

Non-Blocking Send-Receive Diagram



Example: shift messages around a ring (part 1 of 2)

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
  int num, rank, size, tag, next, from;
  MPI_Status status1, status2;
  MPI_Request req1, req2;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank);
  MPI_Comm_size( MPI_COMM_WORLD, &size);
  tag = 201;
  next = (rank+1) % size;
  from = (rank + size - 1) % size;
  if (rank == 0) {
    printf("Enter the number of times around the ring: ");
    scanf("%d", &num);

    printf("Process %d sending %d to %d\n", rank, num, next);
    MPI_Isend(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD,&req1);
    MPI_Wait(&req1, &status1);
  }
```

Example: shift messages around a ring (part 2 of 2)

```
do {
   MPI_Irecv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &req2);
   MPI_Wait(&req2, &status2);
   printf("Process %d received %d from process %d\n", rank, num, from);

   if (rank == 0) {
     num--;
     printf("Process 0 decremented number\n");
   }

   printf("Process %d sending %d to %d\n", rank, num, next);
   MPI_Isend(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD, &req1);
   MPI_Wait(&req1, &status1);
 } while (num != 0);

 if (rank == 0) {
   MPI_Irecv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &req2);
   MPI_Wait(&req2, &status2);
 }

 MPI_Finalize();
 return 0;
}
```

Outline

- Distributed memory systems: the evolution of HPC hardware
- Programming distributed memory systems with MPI
  - MPI introduction and core elements
  - Message passing details
  - Collective operations
- Closing comments

Reduction

```
int MPI_Reduce (void* sendbuf,
         void* recvbuf, int count,
         MPI_Datatype datatype, MPI_Op op,
         int root, MPI_Comm comm)
```

- **`MPI_Reduce`** performs specified reduction operation on specified data from all processes in communicator, places result in process “root” only.
- **`MPI_Allreduce`** places result in all processes (avoid unless necessary)

Operation	Function
`MPI_SUM`	Summation
`MPI_PROD`	Product
`MPI_MIN`	Minimum value
`MPI_MINLOC`	Minimum value and location
`MPI_MAX`	Maximum value
`MPI_MAXLOC`	Maximum value and location
`MPI_LAND`	Logical AND

Operation	Function
`MPI_BAND`	Bitwise AND
`MPI_LOR`	Logical OR
`MPI_BOR`	Bitwise OR
`MPI_LXOR`	Logical exclusive OR
`MPI_BXOR`	Bitwise exclusive OR
User-defined	It is possible to define new reduction operations

Pi program in MPI

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;

        for (i=my_id; i<num_steps;  i=i+numprocs)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD) ;
}
```

MPI Pi program performance

Pi program in MPI

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs; double x, pi, step, sum
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD,
        MPI_Comm_Size(MPI_COMM_WORLD, &

        for (i=my_id; i<num_steps; ; i=i+numprocs)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD) ;
}
```

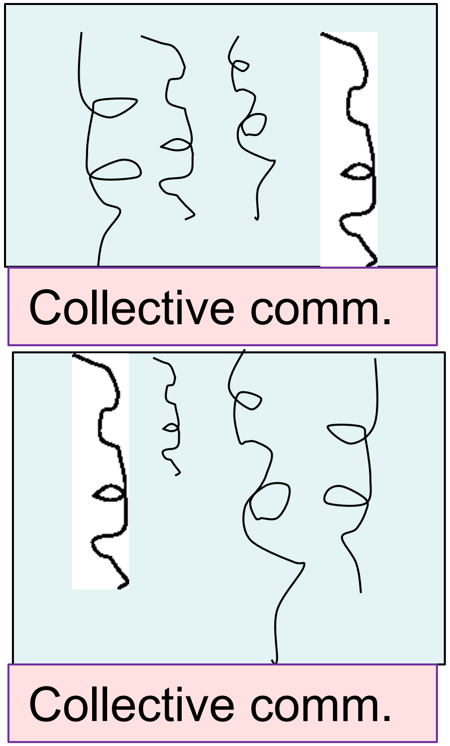
Thread or procs	OpenMP SPMD critical	OpenMP PI Loop	MPI
1	0.85	0.43	0.84
2	0.48	0.23	0.48
3	0.47	0.23	0.46
4	0.46	0.23	0.46

Note: OMP loop used a Blocked loop distribution. The others used a cyclic distribution. Serial .. 0.43.

*Intel compiler (icpc) with –O3 on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Bulk Synchronous Processing

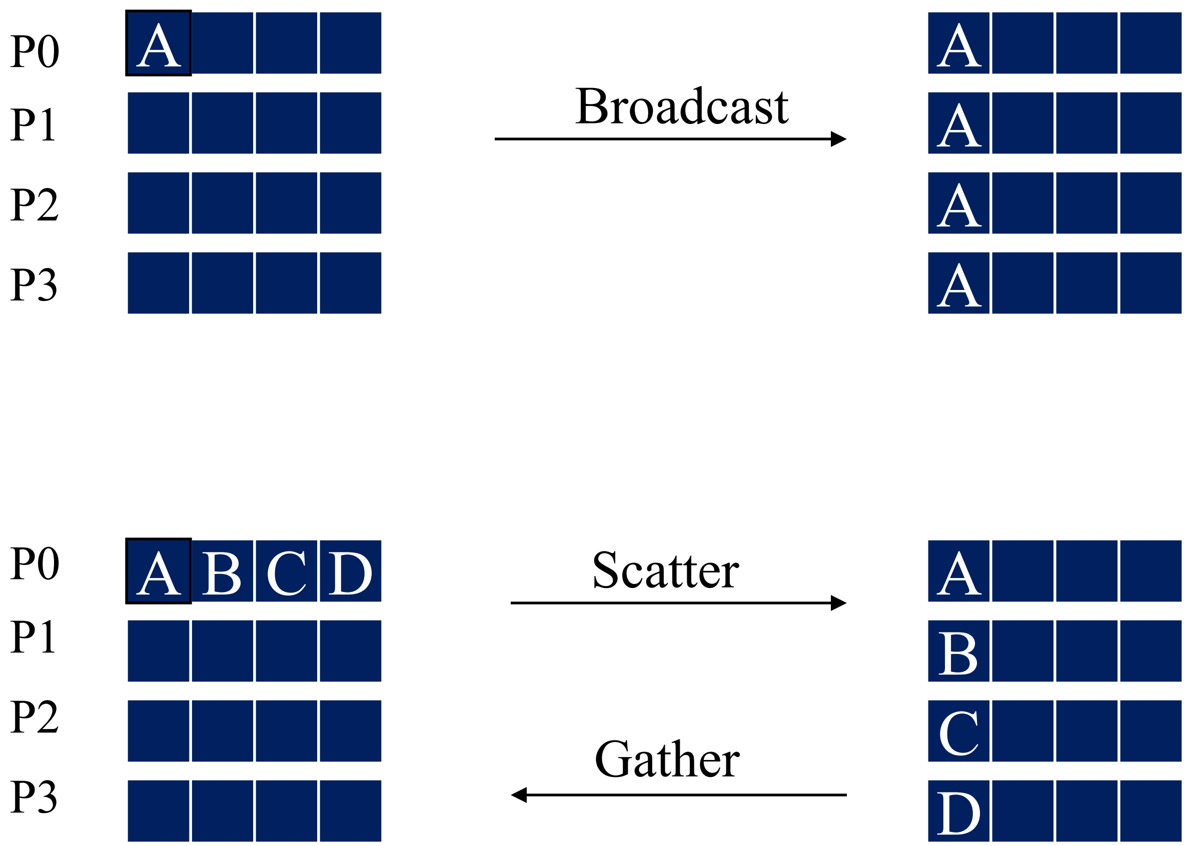
- Many MPI applications have few (if any) sends and receives. They use a design pattern called “**Bulk Synchronous Processing**”.
  - Uses the Single Program Multiple Data pattern
  - Each process maintains a local view of the global data
  - A problem broken down into phases each composed of two subphases:
    - Compute on local view of data
    - Communicate to update global view on all processes (collective communication).
  - Continue phases until complete



MPI Collective Routines

- Collective communications: called by all processes in the group to create a global result and share with all participating processes.
  - **`Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv`**
- Notes:
  - **`Allreduce, Reduce, Reduce_scatter`**, and **`Scan`** use the same set of built-in or user-defined combiner functions.
  - Routines with the “**`All`**” prefix deliver results to all participating processes
  - Routines with the “**`v`**” suffix allow chunks to have different sizes
- Global synchronization is available in MPI
  - **`MPI_Barrier( comm )`**
- Blocks until all processes in the group of the communicator **`comm`** call it.

Collective Data Movement



More Collective Data Movement

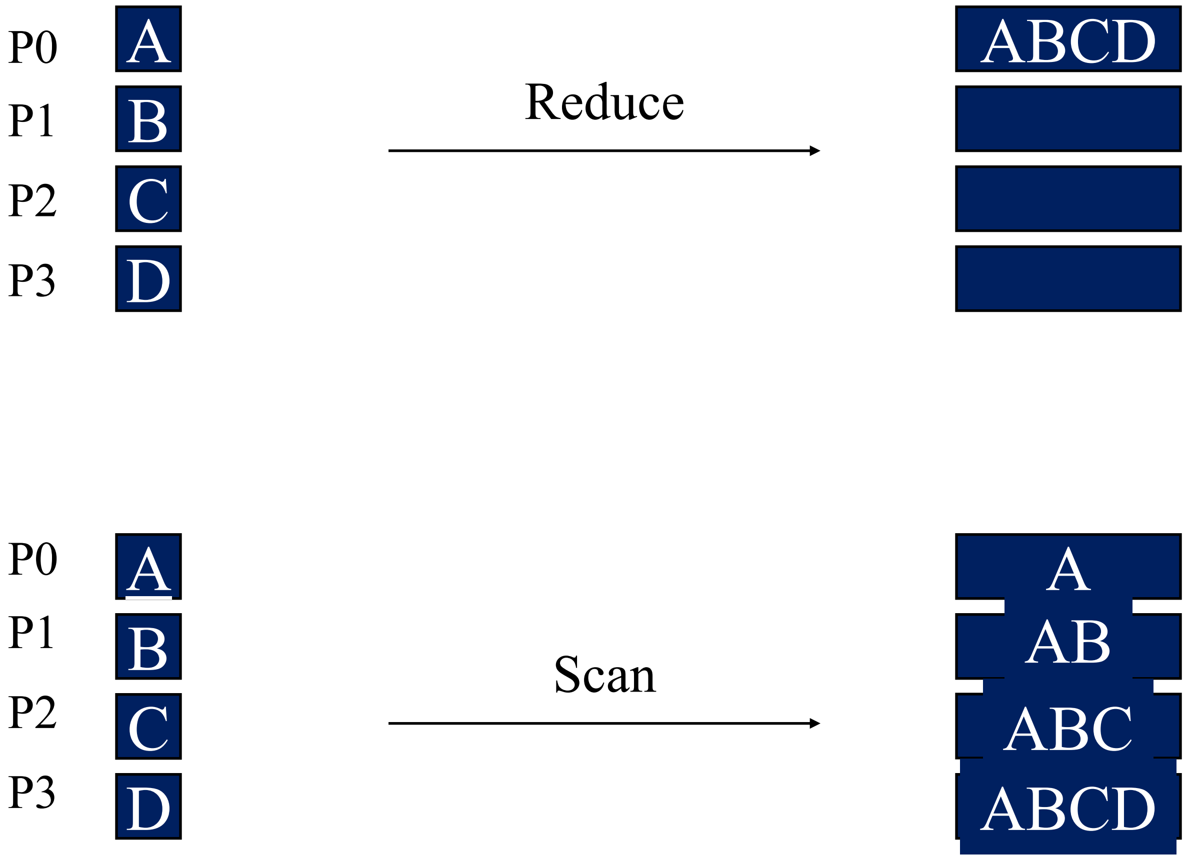
Allgather

P0	A				→	A	B	C	D
P1	B				→	A	B	C	D
P2	C				→	A	B	C	D
P3	D				→	A	B	C	D

Alltoall

P0	A0	A1	A2	A3	→	A0	B0	C0	D0
P1	B0	B1	B2	B3	→	A1	B1	C1	D1
P2	C0	C1	C2	C3	→	A2	B2	C2	D2
P3	D0	D1	D2	D3	→	A3	B3	C3	D3

Collective Computation



Outline

- Distributed memory systems: the evolution of HPC hardware
- Programming distributed memory systems with MPI
  - MPI introduction and core elements
  - Message passing details
  - Collective operations
- Closing comments

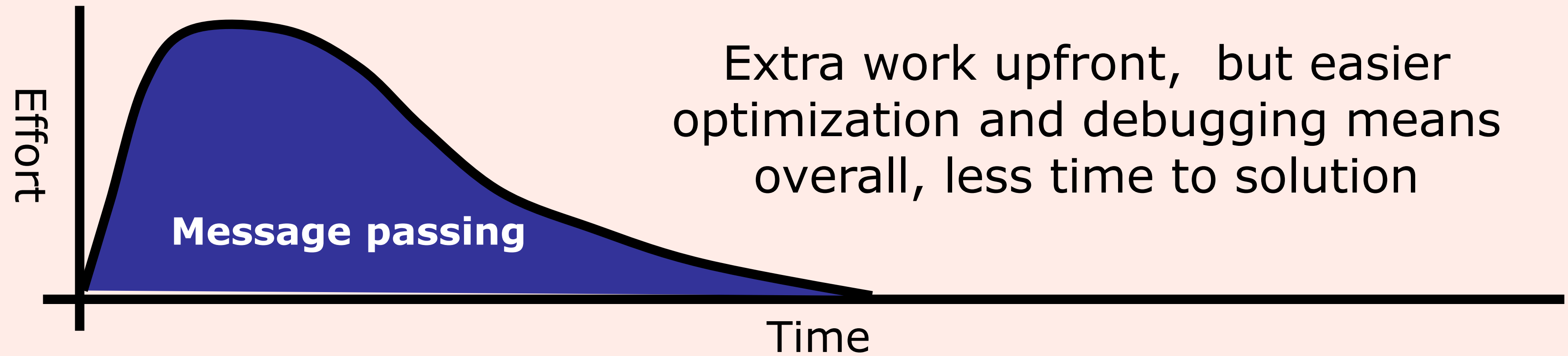
MPI topics we did Not Cover

- Topologies: map a communicator onto, say, a 3D Cartesian processor grid
  - Implementation can provide ideal logical to physical mapping
- Rich set of I/O functions: individual, collective, blocking and non-blocking
  - Collective I/O can lead to many small requests being merged for more efficient I/O
- One-sided communication: puts and gets with various synchronization schemes
  - Implementations not well-optimized and rarely used
  - Redesign of interface is underway
- Task creation and destruction: change number of tasks during a run
  - Few implementations available

MPI isn't as hard as many belive …

- There are over 330 functions in the MPI spec, but most programs only use a small subset:
  - Point-to-point communication
    - **MPI_Irecv, MPI_Isend, MPI_Wait, MPI_Send, MPI_Recv**
  - Startup
    - **MPI_Init, MPI_Finalize**
  - Information on the processes
    - **MPI_Comm_rank, MPI_Comm_size,**
  - Collective communication
    - **MPI_Allreduce, MPI_Bcast, MPI_Allgather**

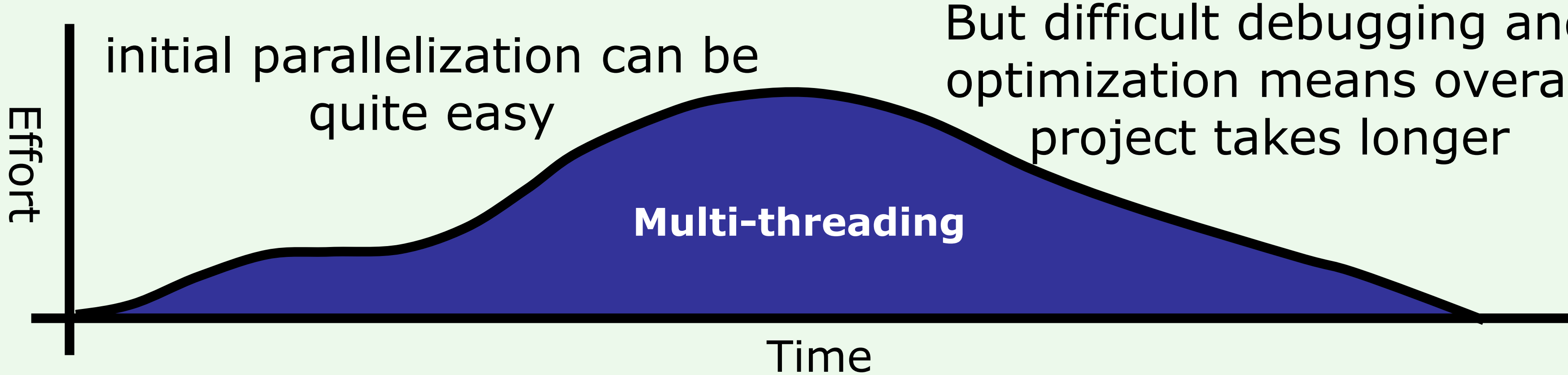
Isn't message passing much harder than multithreading?



Extra work upfront, but easier optimization and debugging means overall, less time to solution

initial parallelization can be quite easy

But difficult debugging and optimization means overall project takes longer



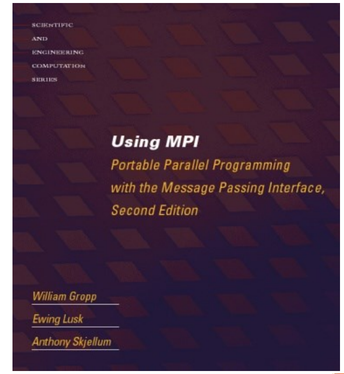
Proving that a shared address space program using semaphores is race free is an NP-complete problem*

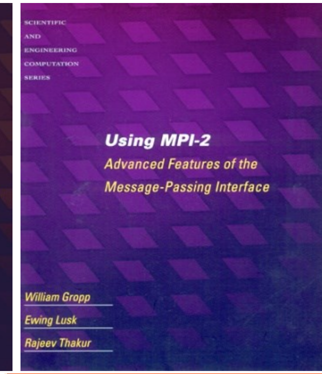
*P. N. Klein, H. Lu, and R. H. B. Netzer, Detecting Race Conditions in Parallel Programs that Use Semaphores, Algorithmica, vol. 35 pp. 321–345, 2003

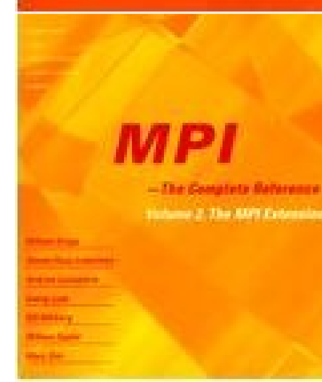
MPI References

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

Books on MPI







- *Using MPI: Portable Parallel Programming with the Message-Passing Interface (2nd edition)*, by Gropp, Lusk, and Skjellum, MIT Press, 1999.
- *Using MPI-2: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Thakur, MIT Press, 1999.
- *MPI: The Complete Reference - Vol 1 The MPI Core,* by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1998.
- *MPI: The Complete Reference - Vol 2 The MPI Extensions*, by Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir, MIT Press, 1998.
- *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
- *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.

Slide source: Bill Gropp, ANL

Backup

- The key constructs of MPI
  - MPI_Init() and MPI_Finalize()
  - MPI_Comm_rize() and MPI_Comm_rank()
  - MPI_Send() and MPI_Recv()
  - MPI_Isend(), MPI_Irecv(), and MPI_Wait()
  - MPI_Bcast(), MPI_Reduce(), MPI_Gather(), and MPI_Scatter()
  - MPI_Barrier()

To do: I need a page for each one of these similar to the one I have now for MPI_send and MPI_Recv

Blocking Send and Receive

```
int MPI_Send (void* buf, int count,
      MPI_Datatype datatype, int dest,
      int tag, MPI_Comm comm)

int MPI_Recv (void* buf, int count,
      MPI_Datatype datatype, int source,
      int tag, MPI_Comm comm,
      MPI_Status* status)
```

- **`MPI_Send`** performs a blocking send of the specified data (“count” copies of type “datatype,” stored in “buf”) to the specified destination (rank “dest” within communicator “comm”), with message ID “tag”

- **`MPI_Recv`** performs a blocking receive of specified data from specified source whose parameters match the send; information about transfer is stored in “status”

By “blocking” we mean the functions return as soon as the buffer, “buf”, can be safely used.

Non-Blocking Comminication

- Non-blocking operations return immediately and pass “request handles” that can be waited on and queried
  - **MPI_ISEND(start, count, datatype, dest, tag, comm, request)**
  - **MPI_IRECV(start, count, datatype, src, tag, comm, request)**
  - **MPI_WAIT(request, status)**
- One can also test without waiting using MPI_TEST
  - **MPI_TEST(request, flag, status)**
- Anywhere you use MPI_Send or MPI_Recv, you can use the pair of MPI_Isend/MPI_Wait or MPI_Irecv/MPI_Wait

Non-blocking operations are extremely important … they allow you to overlap computation and communication.

Launching and closing MPI

- These functions "bracket" every MPI program

`int MPI_Init (int* argc, char* argv[])`

- Initializes the MPI library … called before any other MPI functions.
- agrc and argv are the command line args passed from main()

`int MPI_Finalize (void)`

- Frees memory allocated by the MPI library … close every MPI program with a call to MPI_Finalize

Understanding the process group

- SPMD pattern: use the ID of each process and the size of the process group to choose the data manipulated or the branching through the program

```
int MPI_Comm_size (MPI_Comm comm, int* size)
```

- **`MPI_Comm`**, an *opaque data type,* **a** communication context. Default context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_size`** returns the number of processes in the process group associated with the communicator

```
int MPI_Comm_rank (MPI_Comm comm, int* rank)
```

- **`MPI_Comm`**, an *opaque data type,* **a** communication context. Default context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_rank`** An integer ranging from 0 to “(num of procs)-1”